Generación automática de eventos de prueba para sistemas de IoT*

Lorena Gutiérrez Madroñal, Inmaculada Medina Bulo, and Juan José Domínguez Jiménez

Grupo de Investigación UCASE de Ingeniería del Software.
Universidad de Cádiz
{lorena.gutierrez,inmaculada.medina,juanjosedominguez}@uca.es

Resumen La aplicación en diversas áreas de Internet de las Cosas (IoT) ha ido en aumento en los últimos años. Uno de los principales inconvenientes que tienen los sistemas IoT es la cantidad de información que tienen que manejar. Esta información llega en forma de eventos, cuyo receptor ha de tomar las decisiones correctas, en tiempo real, según los datos recibidos. Viendo la relevancia que tiene el procesado de esta información, resulta fundamental analizar y probar los sistemas IoT que van a trabajar con ella. Para probar las distintas funcionalidades de los sistemas IoT, se necesita una gran cantidad de eventos con estructuras y valores específicos. Conseguir estos eventos de forma manual puede ser una tarea muy costosa y propensa a errores. En este trabajo se presenta un método para la generación automática de eventos de prueba para sistemas de IoT. Los resultados obtenidos en los casos de prueba utilizados muestran su viabilidad.

1. Introducción

Hoy en día la información se ha vuelto vital para tomar decisiones. La información es especialmente crucial en sectores comerciales, para detectar fraudes, para manejar datos de mercado, las redes de sensores... En el 2008 Haller et al. [1] definieron IoT como "Un mundo donde los objetos físicos que se integran en la red se convierten en participantes activos en los procesos de negocios". En este mundo se maneja, en tiempo real, un gran volumen de información que llega de diversas fuentes en forma de mensajes o eventos. Como consecuencia se han desarrollado diversas herramientas, dispositivos y mecanismos para obtener, procesar y transmitir esta información. Uno de los mayores problemas de estos sistemas es monitorizar la información y responder con la menor latencia. Para intentar solventarlo David Luckham propone el *Procesado de Eventos Complejos* (CEP) [2].

A raíz de trabajar con CEP, Luckham plantea el uso de lenguajes de alto nivel para detectar *patrones de eventos* y *reglas*. Donde los *patrones de eventos*

* Este trabajo ha sido financiado por el proyecto DArDOS TIN2015-65845-C3-3-R del Programa Nacional de Investigación, Desarrollo e Innovación del Ministerio de Economía y Competitividad.

son plantillas en las que se reemplazan sus variables por los valores de los eventos y las *reglas* son, en su conjunto, un método para procesar eventos. Años más tarde Schiefer et al. [3] presentan *Event Processing Language* (EPL), un lenguaje estandarizado para definir patrones de eventos y reglas, el cual se ha ido especializando dando lugar a varios EPLs con diferentes propósitos y características, pero siempre con una cosa en común, todos trabajan con eventos.

Profundizando en los conceptos del mundo IoT, hay que rescatar dos conceptos que han de tenerse en cuenta y que fueron definidos por Dunkel et al. [4]:

- **Instancias de eventos**: Toda situación que requiera una reacción por parte del sistema.
- **Tipos de eventos**: Clasificación de las instancias de evento según sus características, es decir, cada instancia de evento pertenece a un *tipo de evento*.

El concepto de *tipos de eventos* fue expresado con mayor detalle por Luckham en [5], donde exponía que: "Si la actividad que queremos procesar puede representarse por más de un evento, de cada uno se recogerán diferentes atributos. Si el **atributo de un evento** es un componente de la estructura de un evento, un **tipo de evento** se crea dependiendo de los atributos que hemos guardado, por lo tanto, la colección de los atributos de un evento determinan su estructura".

A medida que hemos ahondado en el mundo IoT comprobamos la importancia de los eventos, sus atributos y, como consecuencia, los tipos de eventos. Es por esto por lo que nos resulta esencial analizar el comportamiento de los sistemas IoT que van a trabajar con ellos. Por este motivo, para poder realizar las pruebas necesarias a estos sistemas necesitamos producir situaciones extremas a través de los eventos. El resto del artículo se organiza de la siguiente forma: en la Sección 2 se recogen trabajos relacionados, en la Sección 3 se presenta el método con el que se obtienen eventos para pruebas, método que se prueba en diferentes situaciones presentándose los resultados obtenidos en la Sección 4 con diferentes casos de prueba y finalmente en la Sección 5 las conclusiones y el trabajo futuro.

2. Trabajos relacionados

Comenzamos esta sección resaltando aquellos trabajos que están enfocados en hacer pruebas en aplicaciones CEP, cada uno se centra en un tipo diferente de problema o situación de estos sistemas.

Habibi et al. [6] presentan un procedimiento de seis etapas para testear aplicaciones web que manejan eventos. Las pruebas en este tipo de sistemas, cuyo comportamiento se determina por los eventos, necesitan una gran cantidad de eventos que cubran las diferentes etapas. Su procedimiento tenía tanto ventajas como desventajas, estas últimas causadas por el uso de la prueba de mutaciones, ya que esta incrementaba el tiempo de las pruebas.

El objetivo del sistema de código abierto QCEP-TS y su lenguaje EPTDL [7], es hacer pruebas de las funcionalidades CEP, entre las que se incluyen pruebas de activación por tiempo en un entorno simulado. Ellos consideran que nombrar

de forma adecuada los eventos y sus atributos, ayudan a que sus definiciones sean más comprensibles.

FINCoS [8,9] es una herramienta que intenta solventar algunos problemas de los sistemas CEP: la falta de estándares, sus múltiples dominios de aplicación y las métricas. Sus autores piensan que FINCoS puede ayudar a mejorar los motores CEP, y a desarrollar nuevos benchmarks.

Otros estudios se centran en el benchmarking de sistemas CEP [10,11], donde presentan un benchmark CEPBen, que evalúa los comportamientos funcionales de CEP. CEPBen está implementando sobre la plataforma Esper [12] y ha sido usado para explorar los factores de rendimiento y nuevas métricas en la gestión del rendimiento de los sistemas CEP. Los resultados obtenidos de las pruebas de rendimiento demuestran la influencia de los comportamientos funcionales en el rendimiento del sistema CEP.

Finalmente, un repaso sobre los generadores de eventos existentes nos revela que algunos son de propósito comercial [13,14], otros solo pueden ser aplicados en aplicaciones específicas ofertadas por los autores [15], y otros se centran en la generación de eventos de áreas muy específicas [16,17,18], como condiciones ambientales. Dada la necesidad de información por el "usuario de a pie", los generadores de eventos dieron paso a las plataformas IoT que son más accesibles y fáciles de manejar. Al igual que los generadores de eventos, algunas son públicas [19,20,21,22,23] y otras de propósito comercial [24,25,26,27,28], y aunque la mayoría permiten al usuario gestionar el tipo de actividad a monitorizar, también las hay especializadas en áreas.

3. Método para generar eventos

Los programadores que desean hacer pruebas en sistemas IoT, se encuentran los siguientes problemas:

1. No hay una cantidad de eventos suficiente para pruebas.
2. Se necesitan eventos con valores específicos.
3. Hay que esperar la generación de eventos por parte de la fuente de eventos.

Los tres puntos anteriores son claros problemas para los programadores que quieren probar programas que procesan eventos. Por este motivo proponemos un método para generar de forma automática eventos para pruebas. El comportamiento de este método es similar a las plataformas IoT que existen hoy en día, pero con las siguientes diferencias:

1. El usuario puede elegir (sin límite) la cantidad de eventos a obtener.
2. El usuario define el tipo de evento. Se puede personalizar el tipo de evento adaptándose a las necesidades del programador.
3. El usuario puede generar eventos con valores específicos.
4. El usuario puede obtener de forma inmediata los eventos para las pruebas gracias a la automatización del método.

Como según la plataforma IoT que estemos utilizando, el formato de salida de los eventos difiere, se analizan los formatos usados por las siguientes plataformas:

- Buglabs [22] - formato `JSON`
- Carriots [27] - formatos `XML` y `JSON`
- Grovestreams [24] - formato `JSON`
- Lelylan [20] - formato `JSON`
- Particle [21] - formato `JSON`
- Plotly [26] - formatos `JSON` y `CSV`
- Sensorcloud [28] - formato `CSV`
- ThingSpeak [19] - formatos `JSON`, `CSV` y `XML`
- Xively [25] - formato `JSON`
- Zettajs [23] - formato `JSON`

Teniendo en cuenta los formatos de salida existentes en las plataformas IoT, nuestro método generará los eventos en los tres formatos de salida usados por dichas plataformas, es decir, {`JSON, CSV, XML`}.

El método que se propone se divide en etapas (véase la Figura 1) desarrolladas del siguiente modo: el *tipo de evento* del que se desea generar una cantidad de eventos se define siguiendo unas pautas, esta definición será analizada en la etapa de *Validación* y, si esta es correcta, en la última etapa de *Generación de eventos* se obtendrá la cantidad de eventos solicitada, en el formato de salida indicado.

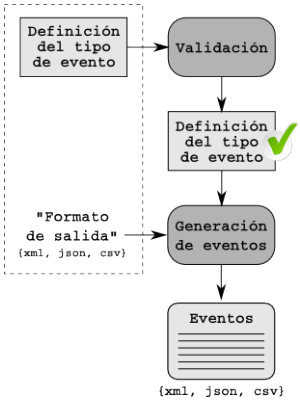

Figura 1. Etapas del método

En las siguientes secciones se detallarán las diferentes etapas del método propuesto para la generación automática de eventos.

3.1. Definición de tipo de evento

El propósito de esta sección es proponer una definición para representar diferentes tipos de eventos. Para intentar uniformar el formato de los eventos,

se necesita que se defina un estándar que unifique la definición de los mismos. En [29] Luckham propone las pautas a seguir por un estándar para la representación de eventos:

- Todos los eventos tienen que ser instancia de un tipo de evento.
- La estructura de los eventos se define por el tipo al que pertenece.
- La estructura se representa como una colección de atributos del evento.

En este mismo trabajo se recomienda el uso de un lenguaje de programación fuertemente tipado (como XML Schema o Java), al igual que se indica que cualquier estándar, a la hora de representar eventos, deberá especificar ciertos datos predefinidos (atributos), como por ejemplo:

- Un identificador único para el evento.
- El tipo de evento.
- La marca de tiempo de la creación del evento.
- La fuente de creación del evento.

Además ha de tenerse en cuenta, que los atributos de un evento pueden ser de tipo simple o de tipo complejo.

Siguiendo las pautas y recomendaciones de Luckham, se escoge XML como lenguaje para definir los tipos de eventos. XML es un lenguaje fuertemente tipado del cual existe una gran cantidad de herramientas y librerías que simplifican su proceso de lectura y análisis.

En la Figura 2 se presentan los diferentes componentes, así como sus propiedades, de la definición de tipo de evento que se propone, los cuales se explicarán, utilizando algunos ejemplos, en los siguientes párrafos.

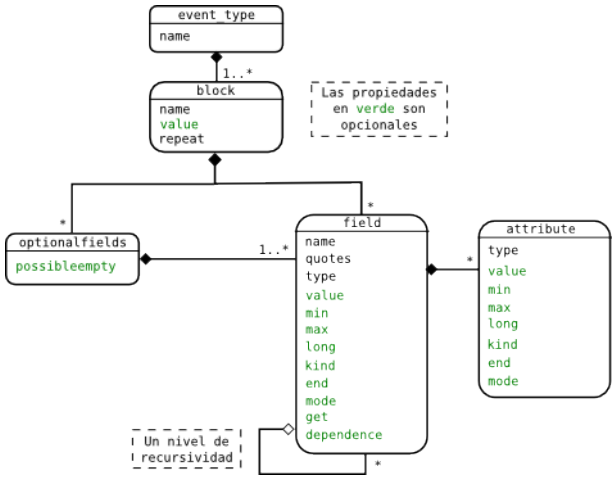

Figura 2. Componentes y propiedades de la definición de tipo de evento.

El componente principal *event_ type*, se compone de *blocks* que se identifican con su propiedad obligatoria `name`. Los *blocks* pueden contener tanto *fields* como *optionalfields*. Los *optionalfields* se componen de *fields*. Los *fields* podrán ser de tipo simple o complejo y tendrán asociados unas propiedades. Si es de tipo complejo, se podrá componer de *attribute* o de *field*; este *field* es un tipo anidado y ya no podrá estar compuesto de *field*.

Dado que el lenguaje utilizado para el estándar es XML, se utilizarán etiquetas y propiedades para definir cada uno de los elementos del tipo de evento. Para que la estructura XML de la definición del tipo de evento pueda ser validada con cualquier validador de este lenguaje de marcado, la primera línea que ha de aparecer es la cabecera con la versión y codificación (véase el Código 1.1). Tras la cabecera, tenemos que indicar el tipo de evento a definir a través de la etiqueta `<event_type>`, donde se indica el tipo de evento a través de la propiedad `name`. En el Código 1.1 se define el tipo de evento `TerminalEvent`.

La etiqueta `<event_type>` se compone de bloques, que se definen con la etiqueta `<block>`. Pueden utilizarse tantos bloques como se quieran, pero uno de ellos tiene que indicar el número de eventos que se desean a través la propiedad `repeat` de la etiqueta `<block>`. Los bloques que no contengan la propiedad `repeat` pueden utilizarse para indicar cualquier tipo de información sobre el tipo de evento. Esta información puede especificarse con la propiedad `value` de la etiqueta `<block>` o a través del componente campo que veremos a continuación (véase el Código 1.1).

Los bloques se componen de campo o campos opcionales, que se definen con las etiquetas `<field>` y `<optionalfields>` respectivamente. Cada campo (`<field>`) define cada uno de los atributos de los que se compone el tipo de evento, y tiene tres propiedades que son obligatorias:

1. Propiedad `name`: contiene el nombre del atributo.
2. Propiedad `type`: se define el tipo del atributo (tipo complejo o simple)
3. Propiedad `quotes`: sus valores son {`true`, `false`}, e indica si los valores del atributo que se define van entrecomillados o no.

En algunos tipos de eventos, ciertos atributos no aparecen siempre; este tipo de atributos se definen con las etiquetas `<optionalfields>`. La única propiedad (opcional) que acepta esta etiqueta es `possibleempty` (valor por defecto `false`), si está a `true` indica que existe la posibilidad de que no aparezca ninguno de esos posibles atributos opcionales, en caso contrario solo aparecerá un atributo. Los componentes de la etiqueta `<optionalfields>` son `<field>` que pueden ser de tipo simple o de tipo complejo. Pueden utilizarse tantos `<optionalfields>` como sean necesarios en la definición del tipo de evento.

En el Código 1.1 se utilizan cada una de las etiquetas y propiedades descritas con anterioridad definiendo el tipo de evento `TerminalEvent`.

Código 1.1. Componentes de la definición del tipo de evento TerminalEvent

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="TerminalEvent">
```

```
  <block name="head" value="Date:2016;Description:...">
  </block>
  <block name="feeds" repeat="150">
    <field name="status" quotes="true" type="ComplexType"...>
    ...
    </field>
    <field name="terminal_id" quotes="false"
           type="Integer" ...></field>
    <field name="timestamp" quotes="false" type="Long" ...>
    </field>
    <optionalfields>
      <field name="responsableId" quotes="true"
             type="Integer" ...></field>
      <field name="destinyId" quotes="true"
             type="String" ...></field>
    </optionalfields>
    <optionalfields possibleempty="true">
      <field name="numluggage" quotes="false"
             type="Integer" ...></field>
    </optionalfields>
  </block>
  <block name="location">
    <field name="lat" quotes="false" type="Float"
           value="36.69"></field>
    <field name="lon" quotes="false" type="Float"
           value="-6.11"></field>
  </block>
</event_type>
```

En el Código 1.1, se representa el tipo de evento `TerminalEvent` que se compone de los atributos: `status`, `terminal_id` y `timestamp` y, como atributos opcionales: `responsableId`, `destinyId` y `numluggage`. El elemento `status` es de tipo complejo: `ComplexStatus` y su valor va entrecomillado, y los elementos `terminal_id` y `timestamp` son de tipo simple: `Integer` y `Long` respectivamente, y sus valores no van entrecomillados.

Según la definición del tipo de evento, las posibles estructuras de los eventos de tipo `TerminalEvent` son:

1. {`status`, `terminal_id`, `timestamp`, `responsableId`}
2. {`status`, `terminal_id`, `timestamp`, `destinyId`}
3. {`status`, `terminal_id`, `timestamp`, `responsableId`, `numluggage`}
4. {`status`, `terminal_id`, `timestamp`, `destinyId`, `numluggage`}

Uno de los atributos opcionales (`responsableId`, `destinyId`) del primer `<optionalfields>` tiene que aparecer en el evento generado. Sin embargo, del segundo `<optionalfields>`, existe la posibilidad de que no aparezca el atributo `numluggage` ya que la propiedad `posibleempty` está definida.

Finalmente, los bloques que no contienen la propiedad `repeat`, se han empleado para que en los eventos se incluya información sobre la fecha y descripción del tipo de evento (bloque `head`), así como la localización (bloque `location`).

3.2. Tipos de datos

Siguiendo las recomendaciones de Luckham, hay que tener en cuenta que los atributos de un evento pueden ser de tipo simple o de tipo complejo. En las siguientes secciones se describirán los tipos de atributo simple y complejo, así como las propiedades de cada uno de ellos que hemos contemplado.

Tipos simples

Los tipos simple de nuestra definición son: enteros (`Integer`), de punto flotante (`Float`), enteros grandes (`Long`), cadenas de caracteres (`String`), alfanuméricos (`Alphanumeric`), booleanos (`Boolean`), fecha (`Date`) y tiempo (`Time`). Para definir un tipo simple se le asigna cualquiera de los identificadores anteriores a la propiedad `type` de `<field>`. En el Código 1.2 se muestra un ejemplo de cada uno de los tipos simples contemplados.

Cada tipo simple contiene unas propiedades específicas que determinan no solo el valor del atributo, sino también su formato. Antes de profundizar en cada una de estas propiedades (la mayoría opcionales), hay que indicar que una propiedad común a todos es `value`. Si esta propiedad tiene asignado un valor este es el que tendrá asignado el atributo definido.

Los tipos enteros, de punto flotante y enteros grandes comparten dos propiedades: `min` y `max`, las cuales definen el rango numérico para el valor del atributo (tienen que estar ambas definidas). Si no están definidas, cada tipo tiene un rango de valores por defecto que será el que adquiera el valor del atributo. El tipo punto flotante tiene la propiedad `long`, que indica el número de decimales que tendrá el valor.

Los tipos cadenas de caracteres y alfanuméricos contienen las propiedades: `long`, `kind` y `end`. La propiedad `long` determina la longitud de la cadena (valor por defecto 10). La propiedad `kind` indica si las letras se escriben en mayúscula (valor por defecto) o en minúscula (se indica asignándole a la propiedad el valor `low`). Si se quiere una cadena de caracteres, o alfanumérica hasta un determinado valor, se puede determinar con el atributo `end`.

El tipo booleano tiene una propiedad: `kind`, que si es igual a `n` los valores generados serán {1, 0} y, en caso de no estar definida los valores serán {true, false}.

Los tipos fecha y tiempo tienen una única propiedad (obligatoria): `mode`, la cual define el formato y los valores de la fecha o del tiempo.

Código 1.2. Ejemplo de tipos simples y sus propiedades

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="TerminalEvent">
  <block name="feeds" repeat="150">
    <field name="terminal_id" quotes="false" type="Integer"
           min="10" max="99"></field>
    <field name="timestamp" quotes="false" type="Long"
           value="2345602942311037"></field>
    <field name="open" quotes="false" type="Boolean" kind="n">
```

```
        </field>
    <field name="open_time" quotes="false" type="Time"
        mode="HH:mm"></field>
    <field name="day" quotes="false" type="Date"
        mode="yy-MM-DD"></field>
    <field name="id_plate" quotes="true" type="String"
        kind="low" long="4"</field>
    <field name="hex_color" quotes="true" type="Alphanumeric"
        long="6" end="F"</field>
    <field name="PoP" quotes="false" type="Float" min="0"
        max="100" long="2"></field>
  </block>
</event_type>
```

En el Código 1.2 se definen los atributos de tipo simple: `terminal_id` (entero), `timestamp` (entero grande), `open` (booleano), `open_time` (tiempo), `day` (fecha), `id_plate` (cadena de caracteres) and `hex_color` (alfanumérico) and `PoP` (de punto flotante).

El atributo `terminal_id` se representa por cualquier número del rango [10, 99], el atributo `timestamp` tendrá el valor de la propiedad `value`, el atributo `open` es booleano y tendrá valores numéricos {1, 0}, los atributos `open_time` y `day` tienen los formatos {horas:minutos} y {año-mes-día} (dos dígitos) respectivamente, `id_plate` es una cadena de caracteres en minúsculas cuya longitud es cuatro, `hex_color` es una cadena alfanumérica con longitud seis cuyo tope es la "F" y `PoP` es de punto flotante con dos decimales y un valor dentro del rango [0, 100].

Tipos complejos

Los tipos complejos se forman combinando cualquiera de los tipos simples y se definen asignándole a la propiedad `type` la palabra clave `ComplexType`. Dentro de los tipos complejos se incluyen los **tipos anidados** con un nivel de anidación, es decir, que un elemento `<field>` podrá tener anidado únicamente otro nivel de elementos `<field>`. En el Código 1.3 se muestran algunos tipos complejos y sus propiedades.

Las propiedades opcionales de los tipos complejos, `get` y `dependence`, afectan al valor del atributo y solamente tienen efecto si están definidas con valor *true*.

Si la propiedad `get` está definida, el comportamiento del tipo complejo a la hora de generar los valores del atributo difiere. En vez de generar valores para todos los hijos del tipo complejo, se escoge aleatoriamente uno de los hijos del tipo complejo para que sea generado.

La definición de la propiedad `dependence` afecta al menos a dos tipos complejos. Si `dependence` se define en un tipo complejo A con valor *true*, significa que al menos un tipo complejo B depende de A. El comportamiento de los tipos dependientes es el siguiente: si del tipo complejo A se escoge el primer hijo, se escogerá el primer hijo de todos los tipos complejos que dependan del tipo complejo A. Dado que la propiedad `dependence` puede aparecer en varios tipos

complejos en la definición del tipo de evento, hay que indicar de qué tipo complejo se depende asignándole a la propiedad `dependence` el nombre del atributo definido como tipo complejo. En resumen, si `dependence` es igual a *true*, otros tipos complejos dependen del tipo complejo donde esta asignación aparece, pero si `dependence` es igual al *nombre de un atributo*, quiere decir que este tipo complejo depende del tipo complejo donde se define ese atributo.

Código 1.3. Ejemplo de tipo complejo y sus propiedades

```
<?xml version="1.0" encoding="UTF-8"?>
<event_type name="TerminalEvent">
  <block name="feeds" repeat="150">
    <field name="terminal_id" quotes="false"
        type="Integer" min="100" max="999"></field>
    <field name="time" quotes="true" type="ComplexType">
     <attribute type="Date" mode="yy-MM-dd"></attribute>
     <attribute type="String" mode="T"></attribute>
     <attribute type="Time" mode="hh:mm"></attribute>
    </field>
    <field name="status" quotes="true" type="ComplexType"
           get="true">
     <attribute type="String" value="OutOfOrder"></attribute>
     <attribute type="String" value="Checkin"></attribute>
     <attribute type="String" value="Cancelled"></attribute>
     <attribute type="String" value="Completed"></attribute>
    </field>
    <field name="departure_loc" quotes="false"
           type="ComplexType" get="true">
     <field name="country_id" quotes="true" type="String"
            long="3"></field>
     <field name="airport_id" quotes="true" type="String"
            long="3"></field>
     </field>
    <field name="destination" quotes="true"
           type="ComplexType" dependence="true">
     <attribute type="String" value="Madrid"></attribute>
     <attribute type="String" value="Rome"></attribute>
     <attribute type="String" value="Paris"></attribute>
     <attribute type="String" value="Berlin"></attribute>
    </field>
    <field name="acronym" quotes="true" type="ComplexType"
           dependence="destination">
     <attribute type="String" value="MDR"></attribute>
     <attribute type="String" value="RME"></attribute>
     <attribute type="String" value="PRS"></attribute>
     <attribute type="String" value="BRL"></attribute>
    </field>
    </block>
</event_type>
```

En el Código 1.3, el tipo complejo *time* se compone de varios tipos simples, un posible valor generado: "16-07-12T12:30". El tipo anidado *departure_loc* contiene los tipos simples: *country_id* y *airport_id*, definidos como campos (con sus correspondientes propiedades). En los tipos complejos *status* y *departure_loc*, se emplea la propiedad `get`, para ambos se escogerá aleatoriamente uno de los hijos para generarse. El atributo *status* tendrá uno de los siguientes valores: {`OutOfOrder, Checkin, Cancelled, Completed`}, y para *departure_loc* se generará un valor para `country_id` o para `airport_id`. Finalmente, el atributo *acronym* depende de *destination*, si se escoge el tercer hijo de *destination*, (`destination:Paris`), se escogerá el tercer hijo de *acronym*, (`acronym:PRS`).

3.3. Etapa de validación

Una vez definido el tipo de evento, se procede a la etapa de *Validación*, en la que se comprueba si la definición del tipo de evento está bien construida de acuerdo a la descripción expuesta en la sección anterior.

En la validación se hace un recorrido por todas las etiquetas que componen la definición del tipo de evento. Se comienza por la etiqueta raíz `<event_type>` comprobando que esta solo contenga etiquetas `<block>` como hijos, que todas contengan la propiedad `name` y que la propiedad `repeat` aparezca una vez.

A continuación comprobaremos los hijos de la etiqueta `<block>` que contiene la propiedad `repeat`; estos tienen que ser `<field>` u `<optionalfields>`. Para la validación de los hijos `<field>` se comprueba que contengan las propiedades obligatorias: `name`, `quotes` y `type`. Para validar los `<optionalfields>`, únicamente se controla que sus hijos sean `<field>`, junto con las anteriores comprobaciones.

Una vez comprobada la estructura del tipo de evento, se validan los tipos de datos: simples y complejos. Para los tipos simples se comprueban las propiedades obligatorias de cada uno de ellos: `name`, `quotes` y `type` para los tipos simples definidos con `<field>` (ya comprobado), y `type` para los tipos simples definidos con `<attribute>`. Además, según el tipo simple (enteros, de punto flotante, enteros grandes, cadenas de caracteres, alfanuméricos, booleanos, fecha y tiempo), controlaremos si las propiedades optativas están bien definidas. Para los tipos complejos se verifica que sus hijos son `<field>` o `<attribute>`, y se validan los tipos de datos simples que lo componen.

Si alguna comprobación anterior no se cumple, se advierte del error.

3.4. Etapa de generación de eventos

El propósito de esta etapa, es generar eventos de manera automática. En esta última etapa de nuestro método se trabaja con el fichero XML validado con la definición del tipo de evento y el formato de salida de los eventos: `JSON`, `CSV` o `XML`. Hay que tener en cuenta que la estructura de los eventos generados variará según el formato de salida indicado, si el formato escogido es `XML` aparecerán `<etiquetas>`, si se desea en formato `JSON` aparecerán {llaves}, o ',' si el formato escogido es `CSV`. La estructura de los eventos generados es algo que se tiene en cuenta a lo largo de todo el proceso de generación.

En esta etapa se sigue un proceso similar a la etapa de *Validación* ya que se comienza por la etiqueta raíz `<event_type>` leyendo cada uno de los hijos `<block>`. Cuando nos encontramos la propiedad `repeat` se profundiza en la etiqueta `<block>` que la contiene leyendo sus hijos `<field>` y/o `<optionalfields>`. Si la definición de un atributo es de tipo simple, se leen las propiedades específicas de dicho tipo y se generará un valor aleatorio cumpliendo las restricciones de las propiedades. En caso de no tener propiedades específicas, el valor generado aleatorio cumplirá las propiedades por defecto del tipo. Si la definición del atributo es de tipo complejo, si están definidas las propiedades `get` o `dependence`, actuaremos en consecuencia. De forma aleatoria cogeremos uno de los hijos del tipo complejo que tenga `get`, y se generará su valor según las propiedades específicas del tipo de dato simple que lo defina. Del mismo modo (aleatoriamente) cogeremos uno de los hijos del tipo complejo que tenga `dependence`, y el de la misma posición de los que dependan de él.

4. Casos de prueba

ThingSpeak [19] es una plataforma de datos abierta y una API para el IoT, que permite a cualquier usuario recoger, almacenar, analizar, visualizar y actuar sobre datos (en forma de eventos) que provienen de sensores o transmisores como Arduino, Raspberry Pi, BeagleBone Black y otro tipo de hardware. Esta plataforma contiene una serie de canales en los cuales existe la posibilidad de compartir dicha información haciéndolos públicos.

Cada día esta plataforma crece, ya que son muchos los usuarios que quieren compartir sus datos a través de estos canales (más de 5000). En cada uno de esos canales se define un tipo de evento (la estructura), que determina el tipo de estudio que se está realizando. Los tipos de evento definidos en estos canales son para estudios meteorológicos, térmicos, referentes a la luz, agua, humedad, ruido, fermentación, conexiones de Internet, etc. Para este trabajo se han analizado los tipos de eventos de más de 450 canales, aproximadamente un 10 % de los canales disponibles en el momento del estudio (como se ha comentado el incremento de canales de la plataforma es considerable). El listado de los canales analizados de la plataforma "ThingSpeak" se encuentra localizado en el repositorio de UCASE[1].

En la Tabla 1 se recogen algunos canales de ThingSpeak estudiados, los eventos que lanza el correspondiente canal en uno de los formatos {JSON, XML, CSV}, su definición en XML y los eventos generados a través de nuestro método.

Cada uno de los datos que aparecen en la Tabla 1, son enlaces que nos llevan a la información sobre el canal en sí (Canal), el conjunto de eventos lanzado actualmente por el canal (Eventos en ThingSpeak), el fichero XML con la definición del tipo de evento (Definición) y el conjunto de eventos generados a través de nuestro método, cantidad especificada en la tabla (Generados).

Para la generación de eventos en los diferentes formatos se han escogido: 4 en JSON, 3 en XML y 3 en CSV. Los resultados demuestran que no hay diferencias

[1] `https://neptuno.uca.es/redmine/projects/sources-fm/repository/show/trunk/src/IoT-EG/test/ChannelsThingspeak`

Canal	Eventos en ThingSpeak	Definición	Generados
11467	Formato JSON	Office Environment	100
21362	Formato JSON	Tiraspol, MD	500
3	Formato JSON	ioBridge Server	1000
41393	Formato XML	IOT House - Basement	100
5186	Formato XML	Channel 5186	500
65409	Formato XML	Pella Weather	1000
77777	Formato CSV	ArduinoProject	100
8557	Formato CSV	Fibaro HC2 - Temperatura	500
9960	Formato CSV	Raspberry Pi - Apartment	1000
1417	Formato JSON	CheerLights	250

Tabla 1. Tipos de eventos de ThingSpeak generados a través del método

entre los eventos generados por el método y los de la plataforma. Con el método propuesto se consigue una cantidad de eventos determinada por el usuario, y eventos con valores específicos, todo de forma automática y personalizada.

5. Conclusiones y Trabajo futuro

Hacer pruebas en sistemas IoT, los cuales consumen y procesan grandes flujos de eventos, es un reto. Consideramos fundamental analizar estos sistemas a través de los eventos que procesan para estudiar su comportamiento en situaciones críticas. Desafortunadamente las investigaciones existentes no están enfocadas en esta dirección. Hoy en día los programadores se pueden encontrar problemas para llevar a cabo este tipo de pruebas: falta de eventos, valores específicos en los eventos, espera de generación de eventos por parte de la fuente.

En este trabajo se propone un método que solventa los problemas anteriores. Éste se divide en etapas: en la primera se define el tipo de evento, en la segunda se valida la definición del tipo de evento y en la tercera se generan los eventos según la definición validada. Este método ha sido probado por más de 450 tipos de eventos reales alojados en la plataforma ThingSpeak, los cuales han servido no solo para validar la funcionalidad del método propuesto, sino también para precisar la definición de tipo de evento en las primeras fases del trabajo.

Nuestro trabajo futuro se centra en la generación de valores más específicos para los eventos que permita mejorar las pruebas a realizar a los sistemas. Por un lado se está implementado una funcionalidad para generar eventos con valores específicos que se obtengan de las consultas EPL que van a manejar dichos eventos. Por otro lado, se plantea la posibilidad de generar valores secuencias en algunos tipos simples de modo que se permita una secuenciación de eventos específicos que permita probar los sistemas.

Referencias

1. Haller, S., Karnouskos, S., Schroth, C.: The internet of things in an enterprise context. Springer (2008)

2. Luckham, D.: The power of events. Volume 204. Addison-Wesley Reading (2002)
3. Schiefer, J., Rozsnyai, S., Rauscher, C., Saurer, G.: Event-driven rules for sensing and responding to business situations. In: Proceedings of the 2007 inaugural international conference on Distributed event-based systems, ACM (2007) 198–205
4. Dunkel, J., Fernández, A., Ortiz, R., Ossowski, S.: Injecting semantics into event-driven architectures. In: ICEIS (1). (2009) 70–75
5. Luckham, D., Schulte, R.: Event processing technical society. Event Processing Glossary—Version **2** (2011)
6. Habibi, E., Mirian-Hosseinabadi, S.H.: Event-driven web application testing based on model-based mutation testing. Information and Software Technology **67** (2015) 159–179
7. Weiss, J., Mandl, P., Schill, A.: Introducing the qcep-testing system for executable acceptance test driven development of complex event processing applications. In: Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation, ACM (2013) 13–18
8. Mendes, M., Bizarro, P., Marques, P.: A framework for performance evaluation of complex event processing systems. In: Proceedings of the second international conference on Distributed event-based systems, ACM (2008) 313–316
9. Mendes, M.R., Bizarro, P., Marques, P.: A performance study of event processing systems. In: Performance Evaluation and Benchmarking. Springer (2009) 221–236
10. Li, C., Berry, R.: Cepben: a benchmark for complex event processing systems. In: Performance Characterization and Benchmarking. Springer (2013) 125–142
11. Li, C.: Performance management of event processing systems. PhD thesis, Aston University (2014)
12. EsperTech: Espertech website. `http://www.espertech.com/esper/index.php`
13. Oy, M.R.F.: Event generator. `http://www.mrf.fi/index.php/timing-system/71-event-generator`
14. Systems, S.: The event generator. `https://www.starcomsystems.com/ru/the-event-generator`
15. Oracle: Introducing the weblogic integration administration console. `https://docs.oracle.com/cd/E13214_01/wli/docs85/manage/intro.html`
16. Dobbs, M.A., Frixione, S., Laenen, E., Tollefson, K., Baer, H., Boos, E., Cox, B., Engel, R., Giele, W., Huston, J., et al.: Les houches guidebook to monte carlo generators for hadron collider physics. arXiv preprint hep-ph/0403045 (2004)
17. Chekanov, S.: Hepsim: a repository with predictions for high-energy physics experiments. Advances in High Energy Physics **2015** (2015)
18. Mangano, M.L., Stelzer, T.J.: Tools for the simulation of hard hadronic collisions. Annu. Rev. Nucl. Part. Sci. **55** (2005) 555–588
19. ThingSpeak: Thingspeak website. `https://thingspeak.com`
20. Lelylan: Lelylan website. `http://www.lelylan.com/`
21. Particle: Particle website. `https://www.particle.io/`
22. Buglabs: Buglabs website. `http://buglabs.net/bugswarm`
23. Zettajs: Zettajs website. `http://www.zettajs.org/`
24. Grovestreams: Grovestreams website. `https://grovestreams.com/index.html`
25. Xively: Xively website. `http://xively.com/`
26. Plotly: Plotly website. `https://plot.ly/`
27. Carriots: Carriots website. `https://www.carriots.com/`
28. Sensorcloud: Sensorcloud website. `http://sensorcloud.com/`
29. Luckham, D.C.: Event processing for business : organizing the real-time enterprise. Hoboken, N.J. John Wiley & Sons (2012)